

0 ソフトウェア工学とは

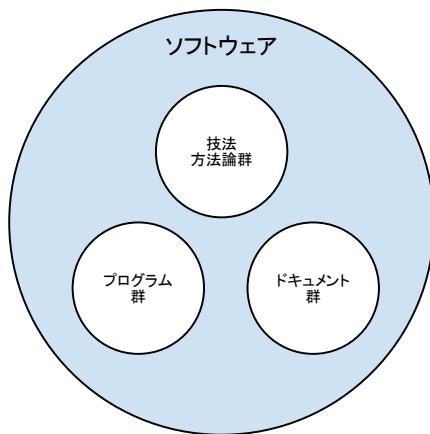
高品質のソフトウェアを効率的・組織的に開発するための手法を研究する学問。

cf. ハードウェアとソフトウェア (pp.1~3)

■ ソフトウェアの定義 (JIS 情報処理用語集 - JIS X0001-1987)

データ処理システムを機能させるためのプログラム、手順、規制、関連文書などを含む知的な創作

cf. p3.fig.1-1



以下のものもソフトウェアに含まれる！

- 要求定義書、外部設計書、内部設計書、流れ図、データベース定義書など
- コーディング規約
- 取扱説明書 (マニュアル)、運用マニュアルなど

(設計の技法、方法論、要因のノウハウ、
経験や技量を含める場合もある)

1 ソフトウェアの概念 (§1.2)

→ 方法論や技法の話

{ 方法論 : 定性的指針、考え方
 { 技法 : 具体的なやり方

■ 分割統治

複雑なものは、分割によって単純な部分に分けて処理 (→分析) する

↓

その結果を統合して全体としての結果を得る

モジュール : 分割された要素 ← ある程度 独立 している

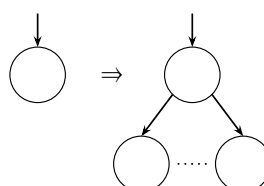
↳ 独立性 ≡ モジュラリティ

□ モジュール分割の方法

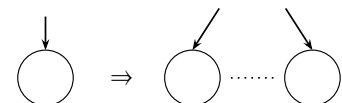
- 深さの分割 (縦の構造)
- 幅の分割 (横の構造)

※ 縦横のバランスが大事

■ 深さの分割



■ 幅の分割



□ モジュール間の関係

- 制御（実行順序）関係 → 構造化技法
- 情報のやり取り → 情報隠蔽



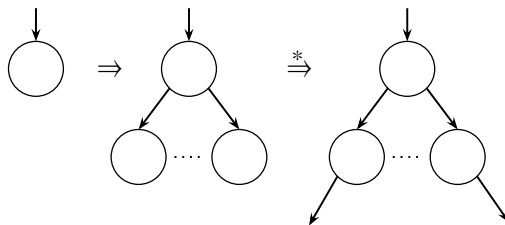
明瞭、単純で「少ない」方が望ましい = 独立性が高い

■ 段階的詳細化

段階的に詳細化する

⋮ ↳ より具体的に、より細かく
 ⋮ (機能を細分化) = 分割 (深さの分割)
 ⋮

詳細化 (≡ 分割) を繰り返す



分析・設計に有効 → **構造化技法**

□ **情報隠蔽** : モジュール間でやり取りする情報を必要最小限に留める (余計な情報は隠す)

□ 抽象化

手続きの抽象化 : 機能を抽象化して内部処理 (実現) を隠す

制御の抽象化 : 制御構造を 基本的な構造 (← 接続・選択・反復) の組み合わせに還元
 構造化プログラミング

データの抽象化 : データとデータに対する操作をセットにする
 → 抽象データ型 → オブジェクト指向

2 ソフトウェアの一般的特性 ①～⑨ (§1.3 : p23～)

- ③ ソフトウェアには潜在的バグが潜んでいる
- ⑨ ソフトウェアの変更には波及効果が生じる

2.1 よいソフトウェアとは (§1.4)

基幹的 (①～⑤)

- 開発コストが低い
- 使い易い (運用コストが低い) : コスト … 時間・人・経費 etc.
- 安全性 (機密保持・障害対策)

処理効率 (①,②)

- ハードウェアリソースを上手く使う (ハードウェアが発達すると重要性が低下)

理解容易性 (①～④)

ソフトウェアがわかりやすい → 使い易い

- 検査しやすい
- 保守しやすい
- ドキュメントが高品質

⇒ ソフトウェアの再利用がしやすい (ソフトウェアの改良、全体部品の再利用を含む)
… 近年重要化

3 ソフトウェア工学の発展経緯 (§2.1 : p33~)

重要性 : ハード > ソフト $\xrightarrow{\text{需要・用途の拡大}}$ ハード $\leq$ ソフト

⇒ ソフトウェア危機 $\left\{ \begin{array}{l} \cdot \text{生産性が上がらない} \\ \cdot \text{品質の相対的低下} \\ \cdot \text{保守コストの増大} \end{array} \right.$ 悪循環

⇒ ソフトウェア工学の提唱

■ ソフトウェア工学

… 理論によって体系付けられた (← 勘や経験に頼る職人技でない) 工学的なソフトウェア開発
→ 種々の理論・方法論・技法が研究開発された

代表的な技法 : 各種の構造化技法
オブジェクト指向技法

各種ツールも発展

ワークベンチ : 統合されたツール群 (IDE)

CASE (Computer Aided System Engineering) ツール

フォワード }
リバース } エンジニアリング
リ }

□ マンマシンインタフェース (ユーザインタフェース) の変化

CUI (文字ベース) $\longrightarrow$ GUI (グラフィックベース)

… $\left\{ \begin{array}{l} \text{キャラクタ端末} \\ \text{キーボード} \end{array} \right.$ … $\left\{ \begin{array}{l} \text{ビットマップディスプレイ} \\ \text{ポインティングデバイス (マウス)} \end{array} \right.$

⇓

$\left. \begin{array}{l} \text{プログラミングスタイルの変化} \\ \text{プログラミング方法論の変化} \\ \text{ソフトウェア開発全体に影響} \\ \text{各種ツールも発達} \end{array} \right\}$ オブジェクト指向

■ ソフトウェア工学の目標と特徴 (§2.2 : p40～)

目標 : 高品質のソフトウェアを効率よく生産・管理する

特徴 ● 工学である (学問である)

× : 経験・直観・職人技

○ : 科学的・理論的・普遍的

- ライフスタイル を視野に入れる

↳ 計画 (構想) → 開発 (テスト) → 運用・保守 → … → 廃棄

←----- ここまで考える ----->

※ トータルコストを抑える

- 大規模ソフトウェアを対象とする

生産プロセスの管理

自動管理を目指す

■ ソフトウェアのプロセスモデル

… ソフトウェアの計画・開発の一連のプロセスモデル

ソフトウェア開発技法と密接な関係

- ウォーターフォールモデル → 構造化技法は主にこれ
- プロトタイピングモデル
- 反復型のモデル (成長モデル、螺旋モデル、インクリメンタルモデル など)
→ オブジェクト指向技法

□ ウォーターフォールモデル

段階的に順に行う各段階で検査して不具合があればやり直し

上流工程 (計画) : 要求定義、システム設計 (外部設計、基本設計)

下流工程 (開発) : プログラミング設計 (内部設計、詳細設計)

プログラミング (コーディング : デバッグを含む)

テスト (単体・結合・システム運用…)

運用・保守

□ プロトタイピングモデル

上流工程で、試作品 (プロトタイプ) を作成・評価をし、必要に応じて上流工程をやり直す

… 上流工程で開発サイクルを回す

□ 反復モデル

一連の開発サイクル (計画開発) を繰り返す (プロセスを進めたり戻したりとも言われる)

イテラティブな開発 : まず作り、それから修正

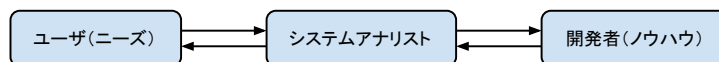
インクリメンタルな開発 : 重要な部分から順に各部を作っていく

4 構造化技法

≪ 工程 ≫		≪ 技法 ≫
要求定義	…	構造化分析
システム設計	…	構造化設計
プログラム設計 プログラミング	…	構造化プログラミング
(コーディング	…	手続き型プログラミング言語 (接続・選択・反復))

■ 要求定義

要求 : ニーズ (ソフトウェアニーズ)
↓ → ユーザが持つ
要求分析 : ユーザのニーズを正確に捉える
↓
要求定義 : ユーザのニーズに答えるシステムを明確・厳密に記述
→ システムのモデル



* システムアナリスト : 双方の意見を聞いて要求定義をまとめる

□ 要求定義の問題点

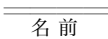
- 要求 (ニーズ) がユーザ自身にとって、明確ではない
- 要求を明確かつ厳密に (分かりやすく) 記述する方法が未確立
- システムアナリスト・ユーザ・開発者の3者の相互理解が困難

→ 時間をかけて相互理解に努め、密にコミュニケーションを図り、極力、明確かつ厳密な要求定義を目指す

■ 構造化分析

- データフローモデルによる要求定義技法
- 主に事務系のシステムに向く（制御系・通信系 には向かない）
↳ リアルタイム処理：有限状態機械が向く
- 道具立て（ツール）
 - － データフローダイアグラム（DFD）
 - － データディクショナリ（DD）
 - － ミニスペック（ミニ仕様書・プロセス仕様書）

□ データフローダイアグラム（DFD） cf. p.86, 表 4-1

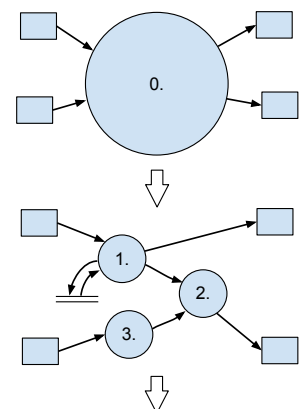
- データの入力元（源泉）・出力先（吸収）：
- データの流れ（データフロー）：
- データの蓄積（ファイル）：
- データの処理・機能（バブル）：

□ 構造化分析の手順

1. コンテキストダイアグラム を作る
↳ 一番最初に作られる最も抽象的な DFD
1つのバブル（システム）＋入力元、出力先
2. コンテキストダイアグラムのバブルを段階的に分割（詳細化）する
3. 各々のバブルが 十分に小さくなる まで分割（詳細化）を繰り返す
↳ 処理内容をミニスペックで記述可能

注意する点

- バブル・データフローには名前をつける
- バブルには固有の番号をつける（cf. p.87, fig4-3）
- データフローは交差させない
- レベル間の整合性を保つ（特に入力元／出力先）※ファイルは例外 →
- データフローを中心に記述「制御」の流れではない
- エラー処理・例外処理は含めない（ミニスペックで記述）
- ファイル周辺のデータの流れに注意



□ データディクショナリ (DD)

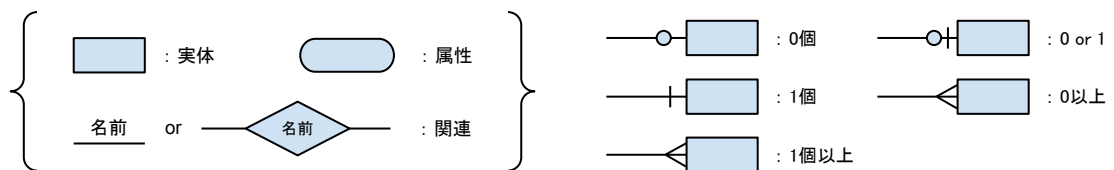
DFD のデータフロー、ファイルの論理構造を記述する。

BNF に類似 (接続・選択・反復) cf. p.90, 表 4-2

- = 定義
- + と (接続)
- [] 1つ選択
- () 選択してもしなくても良い
- { } 繰り返し

DFD の詳細化に合わせて DD も詳細化する (整合性を保つ) ex. p.102

※ データの記述・分析に実体関連図 (ER 図) を用いることがある。 cf. pp.74-76, fig.3-10



□ ミニスペック (ミニ仕様書)

- 最終段階の DFD の各バブルの処理内容を記述
- エラーや例外の処理も記述

※ ユーザも理解できることが重要

* 記述方法

- 構造化言語 (プログラミング言語風自然言語)
- デシジョンテーブル/デシジョンツリー (場合分けの図式)
- フローチャート

■ システム設計 (外部設計、基本設計)

- システムの構造の設計 (論理制御はまだ)
 - ハードウェアシステムの設計
 - * データの流れ → 入出力装置
 - * 使用コンピュータ等のハードウェア構成の決定 (集中/分散、クライアント/サーバ)
 - ソフトウェアシステムの設計
 - * ファイル設計: 入出力データ、共有ファイルの構造、DB など
 - * システム機能設計
 - … 「機能」の構成の決定
 - プログラムの構成 (モジュールへの分割とその全体構成)
 - 「何をするか」
- ※ そのための技法 … 構造化設計

□ システム設計の「外部設計」としての側面

- システムとシステム外との関わりを決定する
 - － 入出力（ハードウェア）の設計
 - － 入出力インタフェース（ソフトウェア）の設計（← ユーザインタフェースを含む）
 - － ファイル（DB）設計

■ 構造化設計

ソフトウェアに要求される機能を段階的に（より単純な機能の組み合わせに）分解していく
 → 分解された機能をモジュールとしてその構成（呼び出し関係、データのやり取り）を定める
 バブルチャートとモジュール構造図を用いる

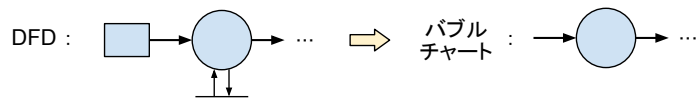
バブルチャート : データ処理の流れを示す

バブル（機能） : 

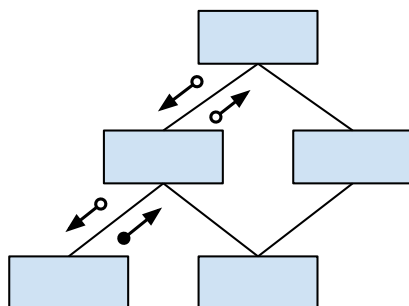
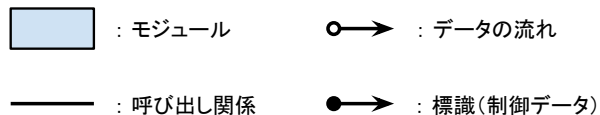
データの流れ（入出力） : 

※ 最終段階の DFD を流用可（多少の変更は可能）

※ ファイル 入力元／出力先は書かない



モジュール構造図 : モジュールの呼び出し関係をデータの流とともに階層的に記述したもの



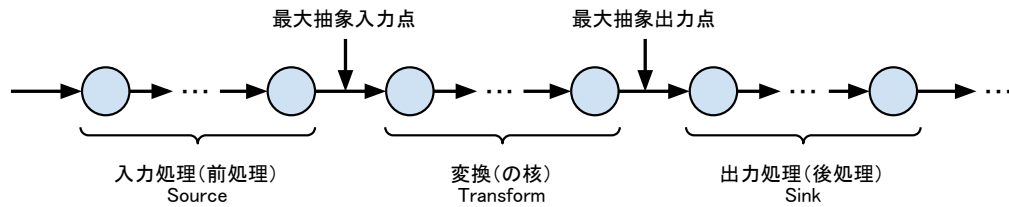
- 上が下を呼び出す
- 呼び出し順が規定されない
順序がある場合は 左 → 右 に並べる

□ 分割

STS 分割 (Source-Transform-Sink)

源泉 変換 吸収

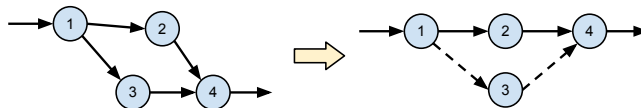
1 本の (主要な) データの流れを S,T,S の 3 つに分割



※ 場合によっては 2 分割 (ST,TS) も可

※ 段階的に繰り返す

※ 一直線でなくても適用できることがある



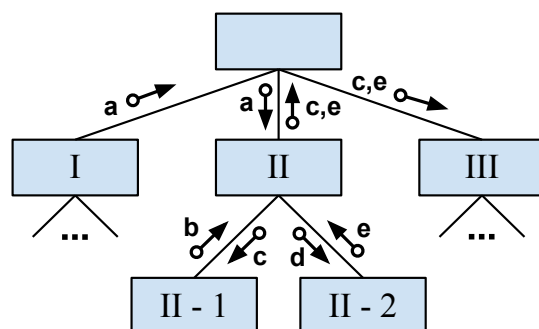
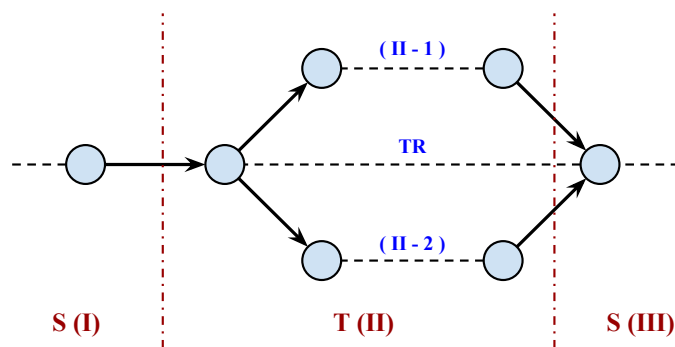
TR 分割 (TRansaction)

データの流れの分岐に対応 (主要なデータの「選択的な」分岐)

[例] バブルチャート

↓ モジュール分割 (STS, TR)

モジュール構造図



□ モジュール分割基準

個々のモジュールの大きさ（適切な大きさ）

- … アルゴリズムが直感的に把握できる
- … 十分なモジュール強度を持つ（しかし、モジュール結合度は大きすぎない）

モジュール間の独立性

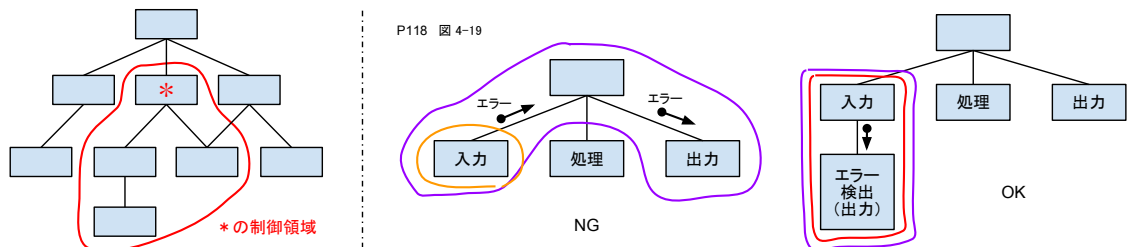
- … モジュール強度
- … モジュール結合度

モジュールの領域

制御領域 : そのモジュールとその下位のモジュール

影響領域 : そのモジュールの実行結果が実行制御に影響を及ぼす範囲

※ 影響領域が制御領域に含まれるのが望ましい



モジュール構造（図）の全体の形状

- … 縦横のバランスがいいのが望ましい
- … モスク型が望ましい（下位モジュールの汎用性が高い）

■ プログラム設計（内部設計・詳細設計）

個々のモジュールの実装

モジュールの $\begin{cases} \text{機能（何をするか）} \\ \text{インタフェース（データの出入り）} \end{cases}$

↓ 段階的詳細化

モジュールの内部処理（ロジック・アルゴリズム、どうやってするか）

※ プログラミング言語の数ステートメントから 1 ステートメント程度まで詳細化

各種技法

* 制御構造に着目した技法

構造化プログラミング（構造化チャート）

* データ構造に着目した技法

ジャクソン法

ワーニエ法

□ 構造化プログラミング（Dijkstra）

* 従来 : フローチャートによるプログラミング（GOTO）

… 「スパゲッティプログラム」になりやすい

↓

プログラムの制御の流れを系統的に整理して（→ 一種の制約 : 接続・選択・反復）

プログラムを見やすく、分かりやすく、作りやすくする

* 考え方（構造化プログラミングの）

● 段階的詳細化

= 実現したい機能に基本パターン（接続・選択・反復）を当てはめて、より単純な機能に分解

※ GOTO の禁止（制御）

参考図書

図解でわかるソフトウェア開発のすべて

Mint 著 日本実業出版社（2000）

ISBN 4-534-03109-2

日経ソフトウェア

<http://itpro.nikkeibp.co.jp/NSW/>

@ IT 情報マネジメント

<http://www.atmarkit.co.jp/im/carc/>